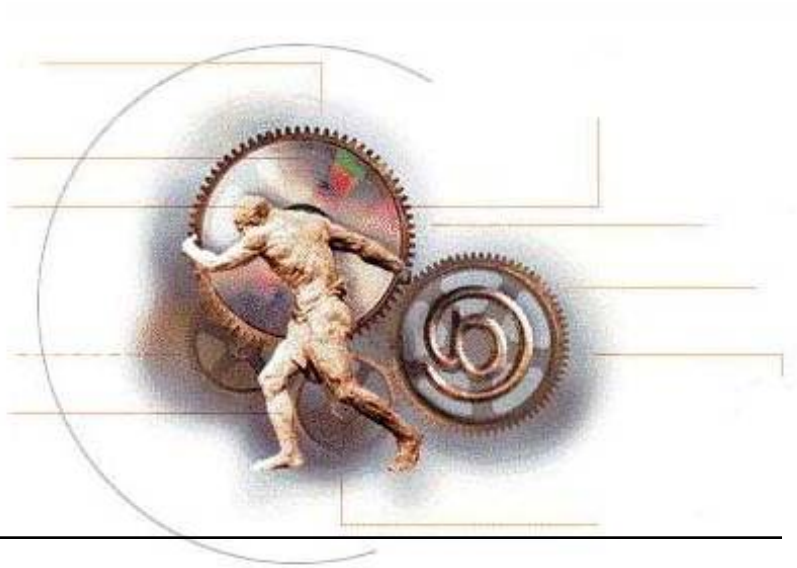


Course 4

搜尋
Search

國立聯合大學 資訊管理學系 陳士杰老師

Outlines

- 本章重點
  - Search
    - 分類觀點
    - Linear Search
    - Binary Search
    - Interpolation Search
    - Hashing

Search 分類觀點

- Internal Search v.s. External Search.
- Static Search v.s. Dynamic Search.
- *Partial Key v.s. Whole Key*
- *Actual Key v.s. Transformation Key*

Internal Search v.s. External Search

- 觀點: 資料量的多寡
- Internal Search:
  - Def: 資料量少，可以一次全部置於**Memory**中進行search之工作
- External Search:
  - Def: 資料量大，無法一次全置於Memory中，須藉助輔助儲存體 (E.g. Disk)，進行分段search之工作
    - B-tree
    - M-way Search tree

Static Search v.s. Dynamic Search

- 被搜尋的資料集合、資料的搜尋範圍、或資料所存在的表格，其內容是否**經常異動** (如: 是否常做資料的插入、刪除或更新) ?
  - 否: **Static**
    - 紙本的字典、電話簿
  - 是: **Dynamic**
    - 日常交易資料、電腦字典

Linear Search (線性搜尋)

- Def:
  - 又稱**Sequential Search**。
  - 自左到右 (或右到左)，逐一比較各個記錄的鍵值與搜尋鍵值是否相同。
  - 若有找到，則Found (成功搜尋); 若Search完整個資料範圍仍未找到，謂之失敗 (Not found)。
- 特質:
  - 檔案記錄不須事先排序
  - 可由**Random Access** (e.g., Array) 或**Sequential Access** (e.g., Link List) 機制支援
  - Time Complexity: **O(n)**，n爲資料個數 (∵線性)

- Linear Search的演算法可分成兩種:
  - Non-Sential (無崗哨) Linear Search
  - Sential Linear Search

Non-Sential Linear Search

Algorithm 1.1: Sequential Search

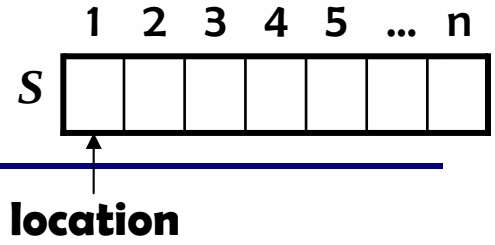
Problem: Is the key *x* in the array *S* of *n* keys?

Inputs (parameters): positive integer *n*, array of keys *S* indexed from 1 to *n*, and a key x.

Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*.)

```
void seqsearch(int n,                          //記錄個數
               const keytype S [ ],            //Array of records (file of records)
               keytype x,                      //欲搜尋的鍵值
               index & location)               //輸出的結果
{
     location = 1;
     while (location <=  n &&  S[location] != x)
       location ++;          ①                ②
     if (location > n)
        location=0;
}
```

- **Found**: location指出記錄的所在位置
- **Not Found**: location重設爲0



分析

- 平均比較次數 (針對 “成功” 的搜尋):

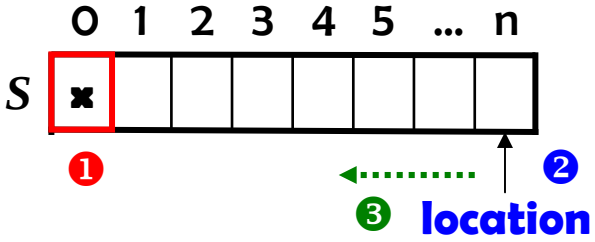
  (1+2+3+...+n)/n

  = n(n+1)/2×1/n = (n+1)/2

  ⇨Time: O(n)

Sential Linear Search

- 觀念: 多一個S[0]記錄，其鍵值設定爲x



Problem: Is the key *x* in the array *S* of *n* keys?
Inputs (parameters): positive integer *n*, array of keys *S* indexed from 1 to *n*, and a key x.
Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*.)

```
void seqsearch (int n,                    //記錄個數
                const keytype S[ ],       //Array of records (file of records)
                keytype x,                //欲搜尋的鍵值
                index & location)         //輸出的結果
```

- **Found**: location表示出記錄的所在位置
- **Not Found**: location爲0

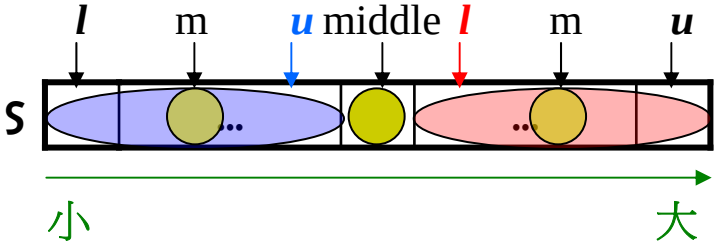
```
{
    ❶ S[0] = x;
    ❷ location = n;
    ❸ while (S[location] != x)
          location --;          ①
}
```

分析

- 以**實際的執行時間**而言:
  - 由於少了 "**測試Search範圍是否結束**" 之比較 (即: location <= n)，所以當n極大時，大約可以省下 **½** 的比較時間。
- 以**Time Complexity分析**而言:
  - 由於仍然是線性搜尋，所以時間複雜度還是 **O(n)**。

Binary Search (二分搜尋)

- 實施前提:
  - 檔案中記錄須事先由小到大排序過
  - 須由**Random (或Direct) access**之機制支援 (e.g., Array)
- 觀念:
  - 每次皆與Search範圍的中間記錄進行比較!!



$$middle = \left\lfloor \frac{l+u}{2} \right\rfloor$$

while ($l \le u$)

$$m = \left\lfloor \frac{l+u}{2} \right\rfloor$$

比較 (k, S[m])

```
    case "=": found, i = m, return i;   //找到了
    case "<": u = m-1;                  //要找的資料在左半部
    case ">": l = m+1;                  //要找的資料在右半部
recurn 0;
```

Algorithm

- Recursion Version:

```
void binsearch (S[], x, low, high)
{
    int   mid;

    if (low < = high){
        mid = ⌊(low + high)/2⌋;
        switch Compare(x, S[mid])
                {
                 Case "=": return middle;
                 Case "<": return binsearch(S, x, middle+1, high);
                 Case ">": return binsearch(S, x, low, middle-1);
                }
     }
     Return -1;
}
```

- Iteration Version:

```
void binsearch (int n,
                const keytype S[],
                keytype x,
                int location)
{
    int low, high, mid;

    low = 1; high = n;
    location = 0;
    while (low <= high && location == 0){
        mid = ⌊(low + high)/2⌋;
        if (x == S[mid])
            location = mid;
        else if (x < S[ mid ])
            high = mid - 1;
        else
            low = mid + 1;
        }
}
```

分析

- 利用Time function

$T(n) = T(n/2) + O(1)$

$= T(n/2) + c$

$= (T(n/4 + c)) + c = T(n/4) + 2c$

$= (T(n/8) + c) + 2c = T(n/8) + 3c$

$= \ldots$

$= T(n/n) + \log_2 n \times c$

$= T(1) + c \log_2 n$ (T(1) = 1, c 爲大於 0 的常數)

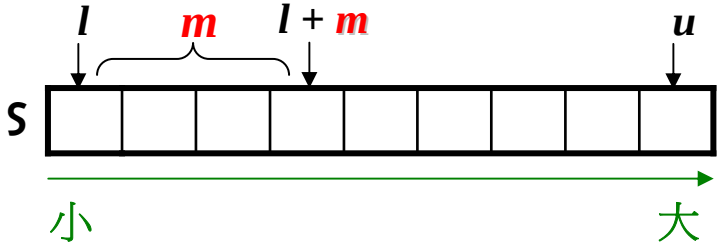
$= 1 + c \log_2 n$

$\therefore T(n) = O(\log_2 n)$

Interpolation Search (插補搜尋)

- 比較符合人類Search之行爲
- 實施前提:
  - 檔案中記錄須事先由小到大排序過
  - 須由**Random (或Direct) access**之機制支援 (e.g., Array)
- 作法:

(m 是一個比較的距離)



$$m = \left\lfloor \frac{x - S[l]}{S[u] - S[l]} \times (u - l + 1) \right\rfloor$$

while ($l \le u$ && i == 0)

$$mid = l + \left\lfloor \frac{x - S[l]}{S[u] - S[l]} \times (u - l + 1) \right\rfloor$$

比較 (x, S[mid])

case ① "=": found, i = mid, return i; //找到了

case ② "<": u = mid-1; //要找的資料在左半部

case ③ ">": l = mid+1; //要找的資料在右半部

recurn 0;

Algorithm

Alogrithm 8.1: Interpolation Search

Problem: Determine whether *x* is in the sorted array *S* of size *n*. Inputs: positive integer *n*, and sorted (nondecreasing order) array of numbers *S* indexed from 1 to *n*.

Outputs: the location *i* of *x* in *S*; 0 if *x* is not in *S*.

```
void interpsrch ( int n,
                  const number S[],
                  number x, index & i)
{
   index low, high, mid;
   number denominator;
   low = 1; high = n; i = 0;
   if  (S[low] ≤ x ≤ S[high])
        while (low <= high && i == 0){
           denominator = S[high] - S[low];
           if (denominator == 0)
               mid = low;
           else
               mid = low + [((x - S[low]) * (high - low)) / denominator];
  Case ① if (x==S[mid])
               i = mid;
  Case ② else if (x < S[mid])
               high = mid - 1;
  Case ③ else
               low = mid + 1;
        }
}
```

} //若S數列中只有一個數字時，防止分母爲0

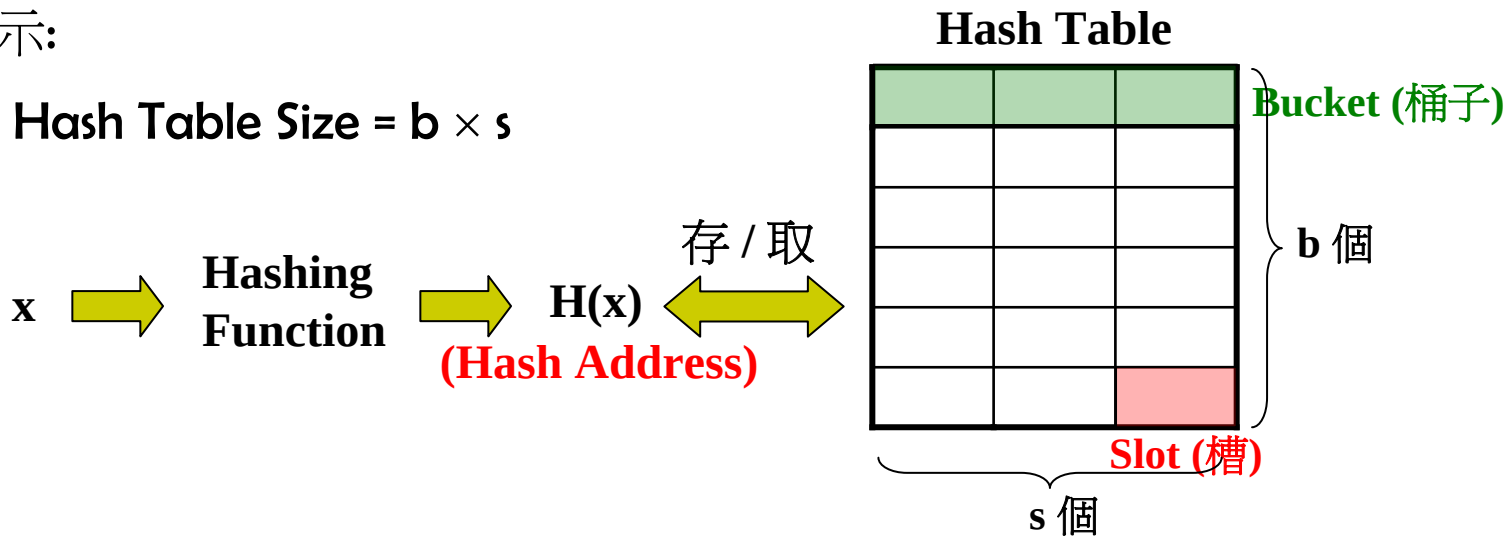
分析

- 其時間分析的效能是與鍵值分佈有關。一般而言，**Uniform Distribution有Best effect.**
- **Time Complexity: $O(\log_2 n) \sim O(n)$**
  - 最佳情況: 同**Binary Search– $O(\log_2 n)$**
    - 每一次都切一半
  - 最差情況: 同線性**Search– $O(n)$**
    - 第一次切割後，會剩下 **(n-1)**; 第二次切割後，會剩下 **(n-2)**筆; **...**依此類推。
    - 即每一次切割後，只有一筆資料被摒除於下一次的搜尋資料之外。

Hashing (雜湊)

- Def: 爲一種資料貯存與搜尋的技術。若要存取某筆資料x，則先將x經過Hashing Function計算，得出Hashing Address，再到Hash Table對應的Bucket中進行存取x的動作。
- Hash Table的結構
  - 由一組Buckets所組成，每個Buckets由一組Slot所組成，每個Slot可存一筆記錄。
  - 圖示:

⇨ Hash Table Size = b × s



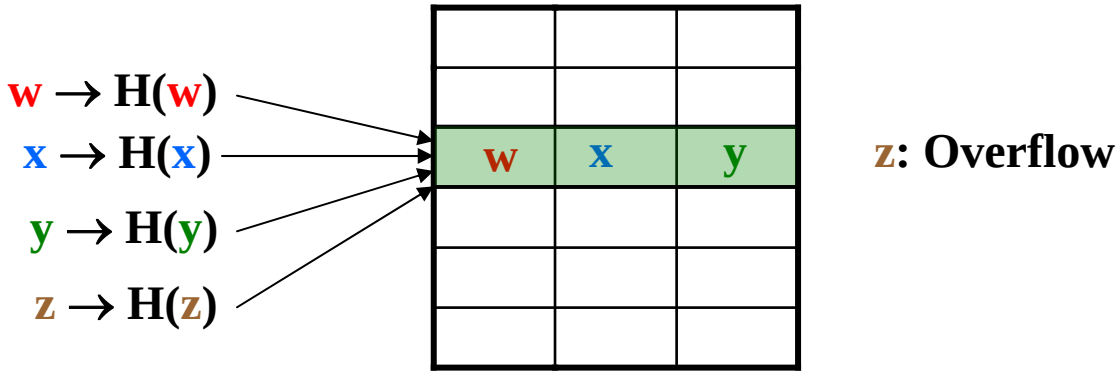
- 優點:
  - 資料搜尋時，記錄不需要事先排序
  - 在沒有**collision**及**overflow**情況下，資料搜尋的**Time**爲 **O(1)**
    - 與資料量 **n** 無關
  - 保密性高
    - ∵若不知**Hashing function**，則無法存取資料
  - 可作資料壓縮之用

相關術語

- **Identifier Density 與 Loading Density**
  - **Def:** 令T爲identifier總數，n爲目前使用者的identifier個數，b爲Hash Table之Bucket數目，S爲Bucket中之Slot數目，則:
    - Identifier Density = n/T
    - Loading Density = n/(b×S) = α
      - α 愈大，則表示Hash Table Utilization高，但相對地Collision / Overflow機率也變高。
- **Collision**
  - **Def:** 不同的資料 (e.g., x與y) 在經由Hashing Function計算，竟得出相同的Hashing Address (即 H(x) = H(y)) 稱之。

Overflow

- Def: 當Collision產生，且Bucket中無多餘的Slot可存資料稱之。



- 有Collision並不一定有Overflow，但有Overflow，則必有Collision發生。
- 若Bucket只有一個Slot，則Collision = Overflow

Hashing Function設計

- 一個良好的Hashing Function須滿足下列三個準則:
  - 計算簡單
  - **Collision** 宜盡量少
    - Ex: x mod 2就是不好的Hashing Function!!
      (∵不是0就是1, 會經常發生Collision)
  - 不要造成**Hashing Table**局部儲存 (局部偏重) 的情況
    - 會引發 "空間利用度差" 與 "Collision上升" 的缺失
- 上述準則導引出兩個名詞:
  - Perfect Hashing Function (完美的雜湊函數)
    - Def: 此Hashing Function 絕對不會有Collision發生
    - 前提: 須先知道所有資料 (for Static Search)
  - Uniform Hashing Function (均勻的雜湊函數)
    - Def: 此種Hashing Function計算所得出的Hashing Address，對應到每個Bucket No.的機率皆相等。(不會有局部偏重的情況)

4種常見的Hashing Function

- Middle Square (平方值取中間位數)
- Mod (餘數，或 Division)
- Folding Addition (折疊相加)
- Digits Analysis (位數值分析)

Middle Square (平方值取中間位數)

- Def: 將Key值取平方，依Hashing Table Bucket數目，選取適當的中間位數值作爲Hash Address。
  - e.g., 假設有1000個Bucket，範圍編號爲000~999，若有一數值x = 8125，試利用Middle Square求其適當之Hash Address
  - Sol:

    x = 8125 ⇨ (取平方) 66015625

    取中間三位 ⇨ 156 = Hash Address (取015亦可)

Mod (餘數，或 Division)

- Def: H(x) = x mod m
- m的選擇之注意事項:
  - m不宜爲 "2"
    - 求得的位址僅有0或1，collision的機會很大
  - m的選擇最好是質數 (除盡1和除盡自已)

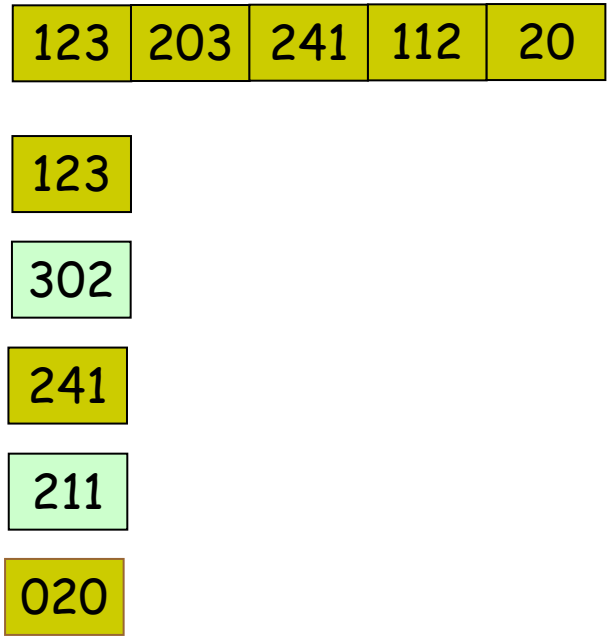
Folding Addition (折疊相加)

- Def: 將資料鍵值切成幾個相同大小的片段，然後將這些片段相加，其總和即爲Hashing Address
- 相加方式有兩種:
  - Shift (移位)
  - Boundary (邊界)
- 若有一資料x = 12320324111220，請利用兩種不同的Folding Addition方法求Hashing Address (假設片段長度爲3)。

- **Sol:**
  - x=12320324111220 are partitioned into three decimal digits long.
    P1 = 123, P2 = 203, P3 = 241, P4 = 112, P5 = 20.
  - Shift folding:

$$h(x) = \sum_{i=1}^{5} P_i = 123 + 203 + 241 + 112 + 20 = 699$$

  - Folding at the boundaries:



h(x) = 123 + 302 + 241 + 211 + 20 = 897

Digits Analysis (位數值分析)

- **Def:** 當資料事先已知，則可以選定基底r，然後分析每個資料之同一位數值。
  - 若很集中，則捨棄該位;
  - 若很分散，則挑選該位，而挑選的位數值集合成 Hashing Address。
- **Ex:**

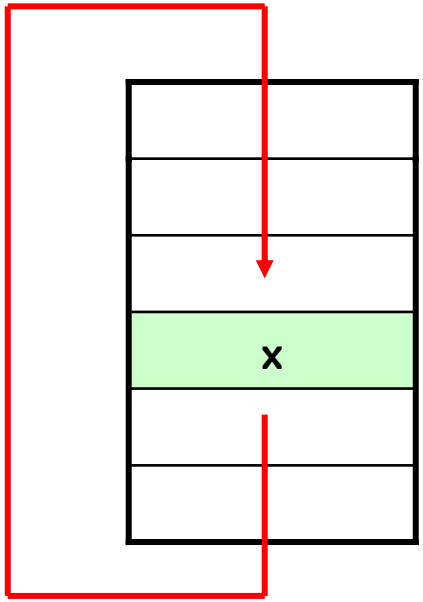
電話號碼	鍵值	位址
02-9848	7 5 8 6	4 5 6
02-9888	7 8 6 4	8 8 4
02-7677	6 7 8 5	7 7 5
02-8921	6 4 3	2 6 3
02-9967	5 8 7	6 5 7
02-8837	4 8 2	3 4 2
02-7847	3 8 1	4 3 1

4種常見的Overflow處理方式

- Linear Probing (線性探測)
- Quadratic Probing (二次方探測)
- Rehashing (再雜湊)
- Link List (鏈結串列，或稱Chain)

Linear Probing (線性探測)

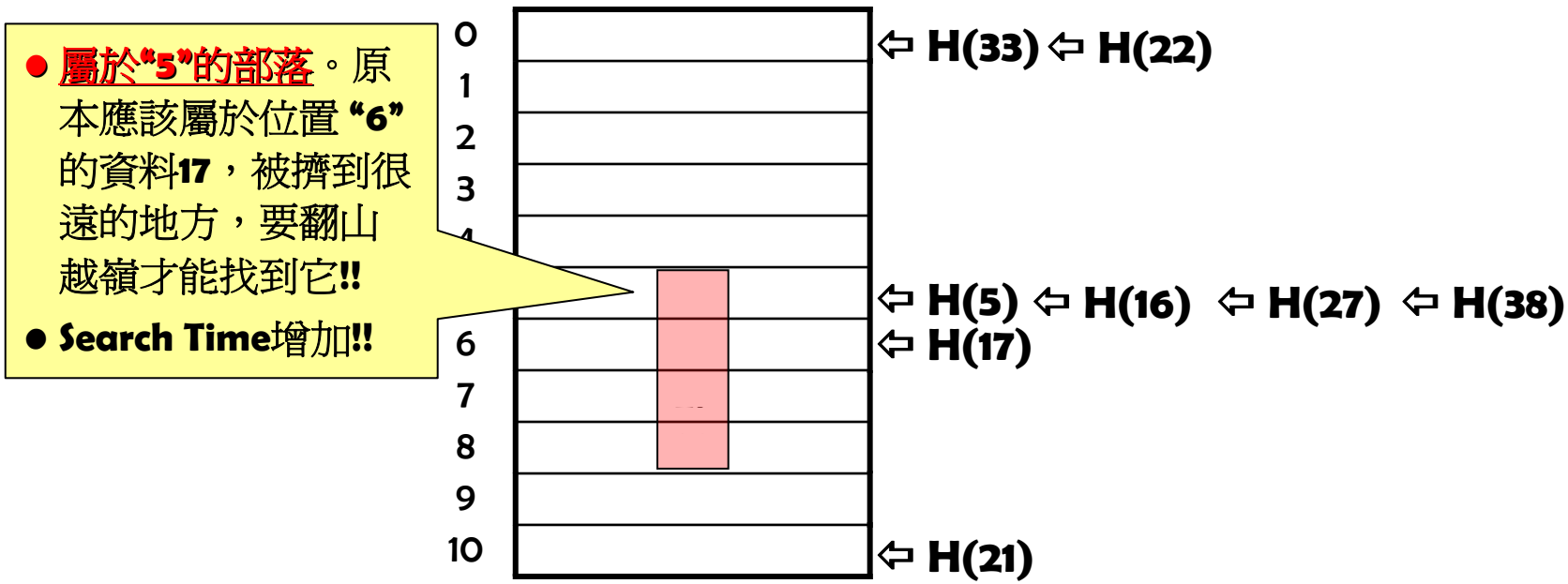
- Def: 又稱Linear Open Addressing。當H(x)發生overflow，則循著H(x)+1, H(x)+2, ..., H(x)-1順序，逐步搜尋，直到:
  - 遇見有空的Bucket
  - 已搜尋完一圈爲止 (表示Hash Table Full，無法store)
- 圖示:



- Hash Table有11個buckets (編號: 0~10)，每個bucket只有一個slot，假設Hashing Function = **x mod 11**，並採取 "Linear Probing"處理overflow。試依照下列資料次序存入Hash Table，會得到什麼結果?

5, 16, 33, 21, 22, 27, 38, 17

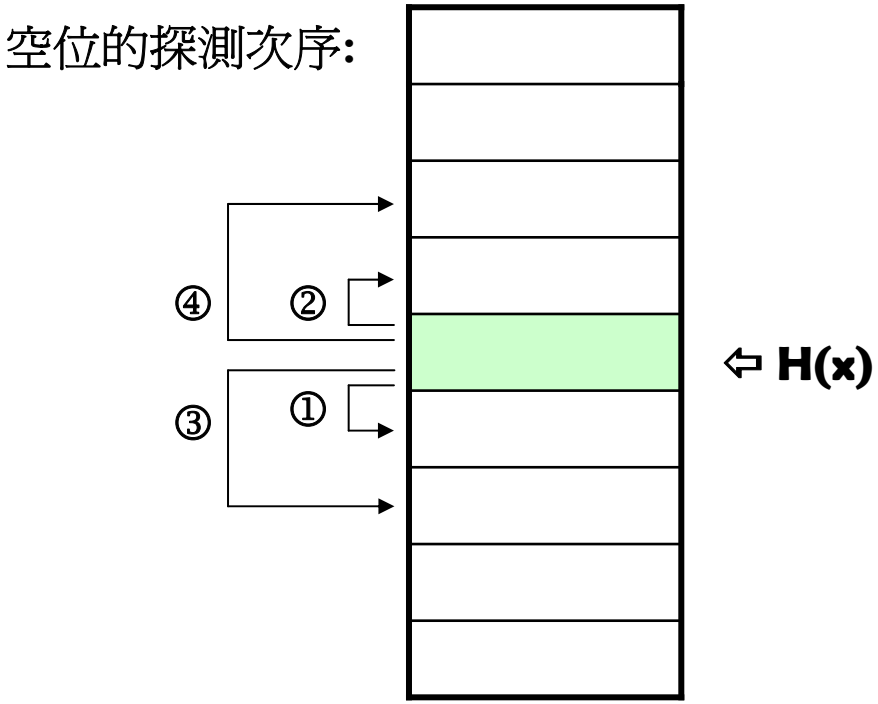
- Sol:



- 缺點: 易形成**資料群聚 (Clustering)**現象，增加Searching Time

Quadratic Probing (二次方探測)

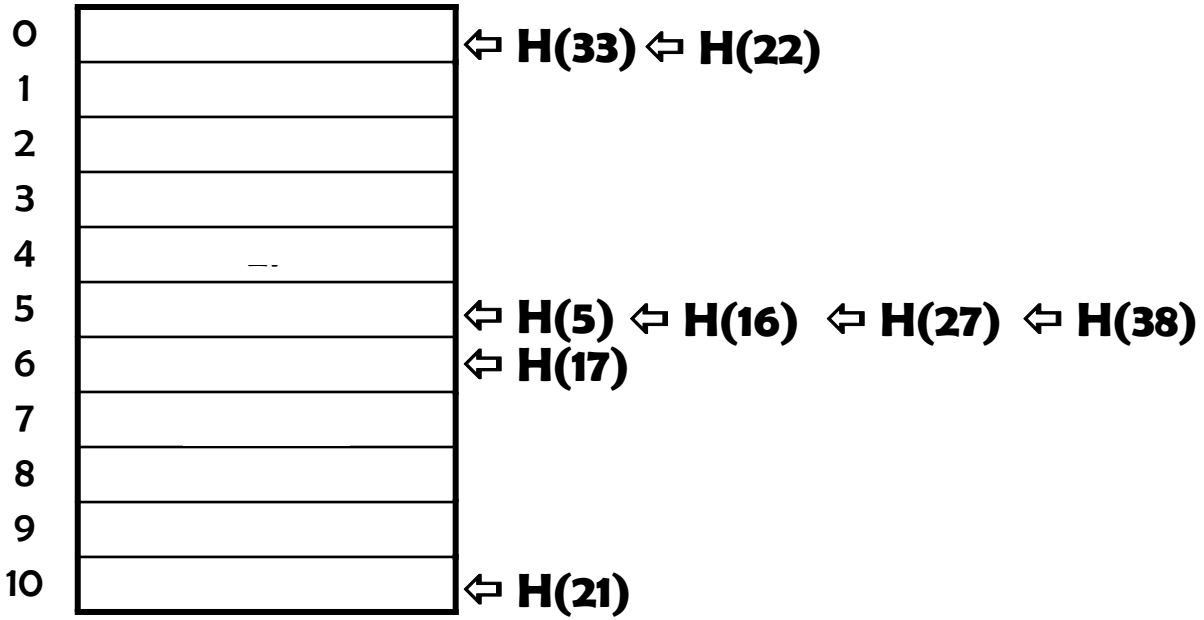
- **Def:** 爲改善**Clustering**現象而提出。當**H(x)**發生**overflow**時，則探測 **(H(x) ± i²) mod b**，**b**爲**bucket**數，**1 ≤ i ≤ (b-1)/2**
- 圖示:



- 承接上題，並改採 “Quadratic Probing”處理overflow。則Hash Table內容爲何?

5, 16, 33, 21, 22, 27, 38, 17

- Sol:



- 承接上題，44 ⇒ ?
- Sol:

H(44) = 0 ⇨ $(0+1^2) \bmod 11 = 1$

⇨ $(0-1^2) \bmod 11 = 10$

⇨ $(0+2^2) \bmod 11 = 4$

⇨ $(0-2^2) \bmod 11 = 7$

⇨ $(0+3^2) \bmod 11 = 9$

⇨ $(0-3^2) \bmod 11 = 2$

負值需先加上**11的適當倍數**，再取**mod!!**

0	33
1	22
2	
3	
4	27
5	5
6	16
7	17
8	
9	38
10	21

Rehashing (再雜湊)

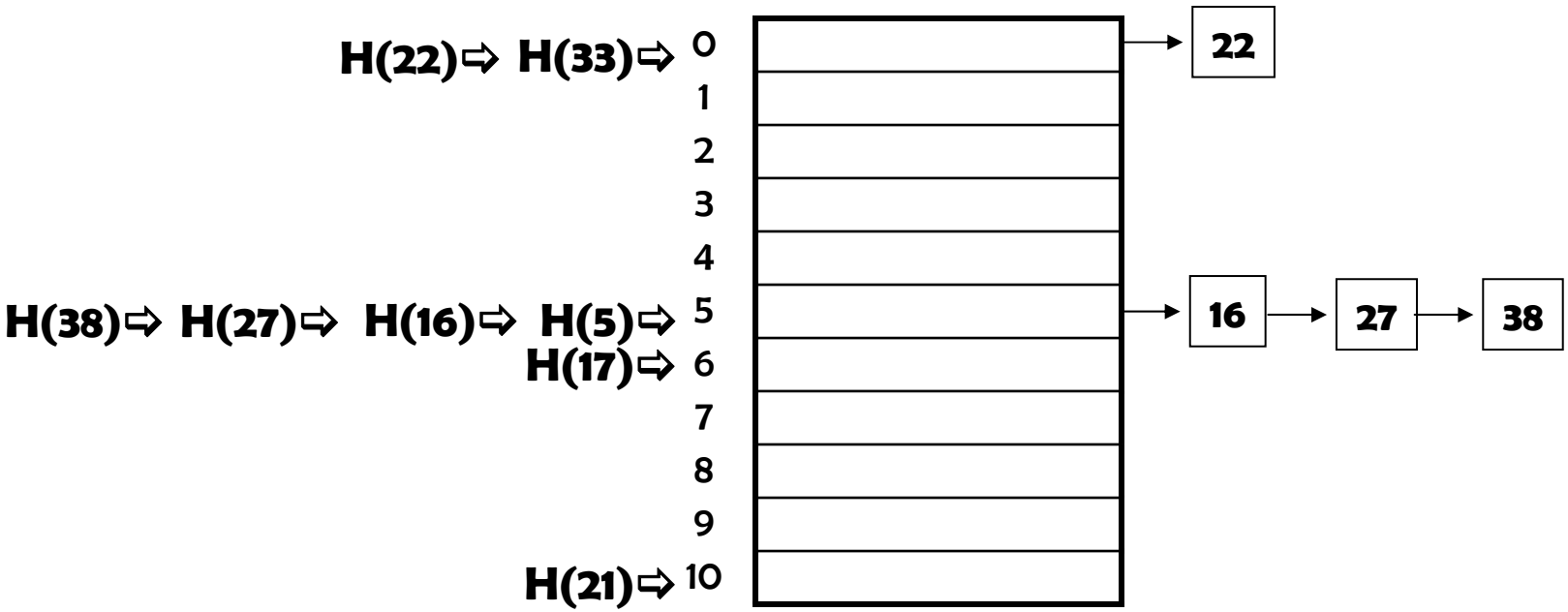
- Def: 提供一系列的Hashing Functions: $f_1, f_2, f_3, \ldots f_n$。若使用 f_1 發生overflow，則改用 f_2; 以此類推，直到:
  - 沒有overflow發生
  - 全部function用完

Link List (鏈結串列，或稱Chain)

- 將具有相同Hashing Address的資料，以**Link list**方式串連在同一Bucket中。
- 承接上題，並改採 "Quadratic Probing"處理overflow。則Hash Table內容爲何?

5, 16, 33, 21, 22, 27, 38, 17

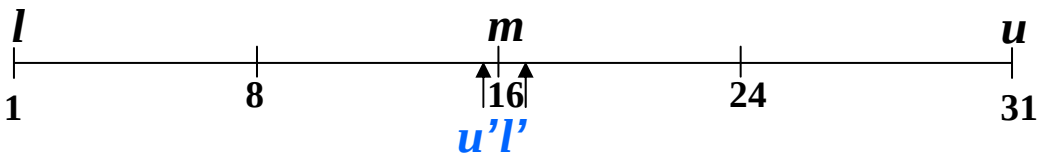
- Sol:

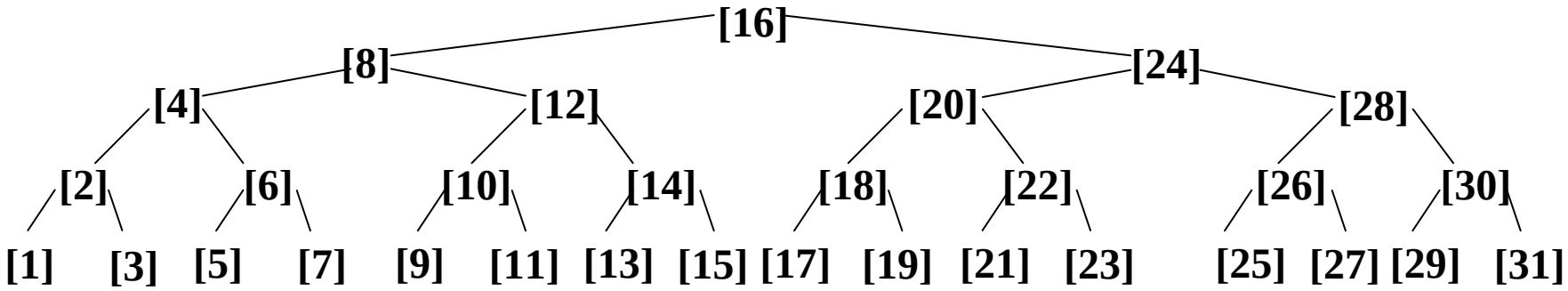


補　充

補 1: Decision Tree for Binary Search

- 目的:
  - 用以描述與了解**Binary Search**的比較行為
  - 一定是二元樹
- 給出n = 31筆記錄之Binary Search的決策樹
- Sol:





- 欲搜尋記錄在第18筆，則比較 __ 次才能找到
- 最多之比較次數為何 (比較幾次後，即知失敗)?___次
- n筆記錄，最多的比較次數 =

※範例練習※

- 繪出n=12筆記錄，執行Binary Search之Decision Tree
- 有下列資料，26, 55, 77, 19, 13, 2, 5, 49
  - 以Binary Search找 “55” 須比較幾次?